Universidad Autónoma de Madrid

excelencia Campus Internacional UAM CSIC+

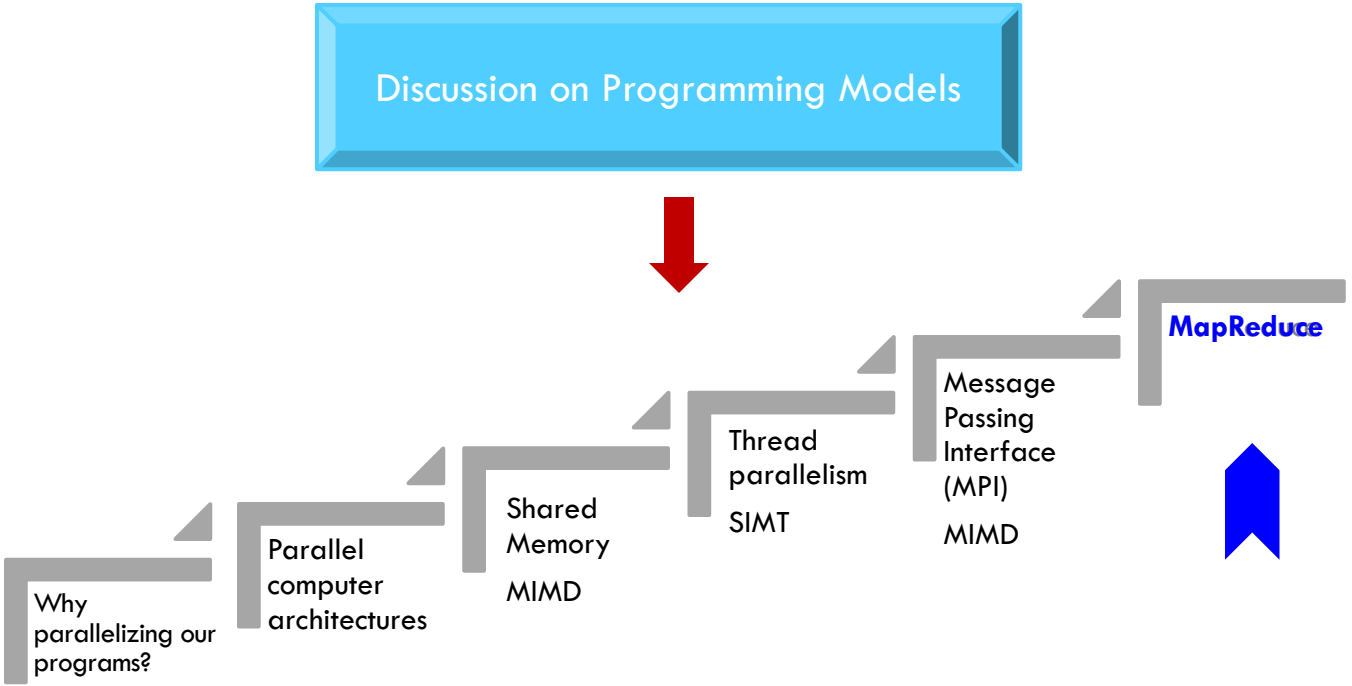
Escuela Politécnica Superior

Modelos de Programación

Procesamiento de Datos a Gran Escala

Francisco J. Gómez Arribas

Objectives


Discussion on Programming Models
Why parallelizing our programs?
Parallel computer architectures
Shared Memory
MIMD
Thread parallelism
SIMT
Message Passing Interface (MPI)
MIMD
MapReduce

MapReduce

- Concepts of MapReduce:
  - Basics
  - MapReduce data flow
  - Additional functionality
  - Scheduling and fault-tolerance in MapReduce

Problem Scope

- *MapReduce* is a programming model for data processing
- The power of MapReduce lies in its ability to scale to 100s or 1000s of computers, each with several processor cores
- How large an amount of work?
  - Web-Scale data on the order of 100s of GBs to TBs or PBs
  - It is likely that the input data set will not fit on a single computer's hard drive
  - Hence, a distributed file system (e.g., Google File System-GFS or Hadoop Distributed File System HDFS) is typically required.

Commodity Clusters

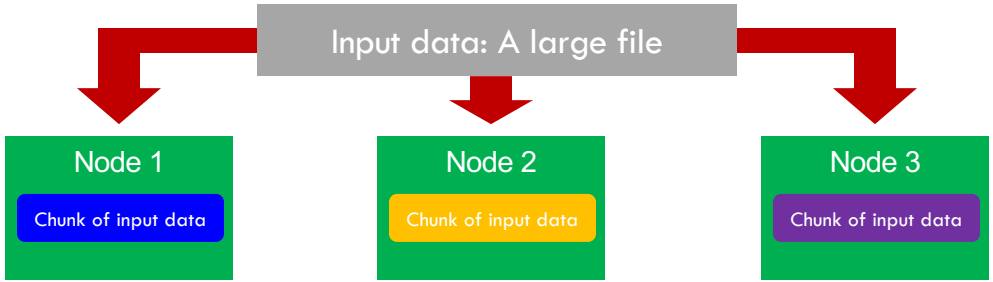
- MapReduce is designed to efficiently process large volumes of data by connecting many commodity computers together to work in parallel
- A *theoretical* 1000-CPU machine would cost a very large amount of money, far more than 1000 single-CPU or 250 quad-core machines
- MapReduce ties smaller and more reasonably priced machines together into a single cost-effective *commodity cluster*

Isolated Tasks

- MapReduce divides the workload into multiple *independent tasks* and schedule them across cluster nodes
- A work performed by each task is done *in isolation* from one another
- The amount of communication which can be performed by tasks is mainly limited for scalability reasons
- The communication overhead required to keep the data on the nodes synchronized at all times would prevent the model from performing reliably and efficiently at large scale

Data Distribution

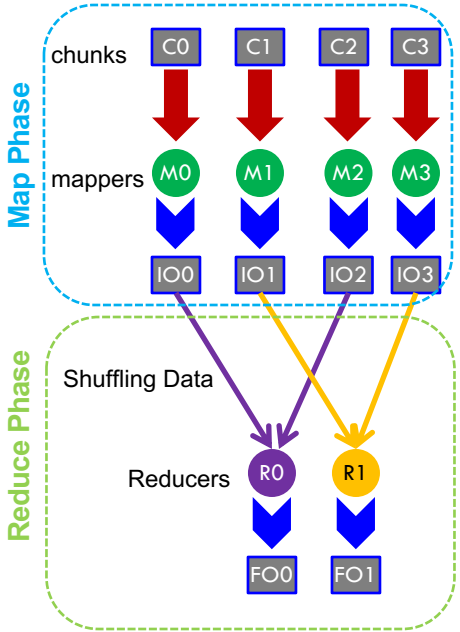
- In a MapReduce cluster, data is distributed to all the nodes of the cluster as it is being loaded in
- An underlying distributed file systems (e.g., HDFS) splits large data files into chunks which are managed by different nodes in the cluster



- Even though the file chunks are distributed across several machines, they form *a single namespace*

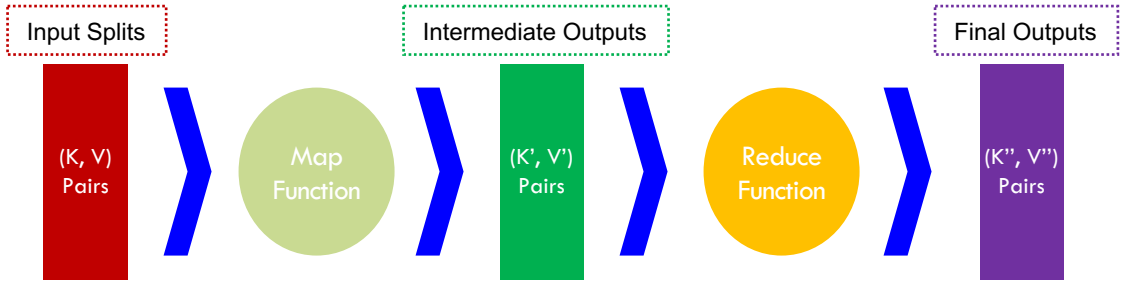
MapReduce: A Bird's-Eye View

- In MapReduce, chunks are processed in isolation by tasks called *Mappers*
- The outputs from the mappers are denoted as intermediate outputs (IOs) and are brought into a second set of tasks called *Reducers*
- The process of bringing together IOs into a set of Reducers is known as *shuffling process*
- The Reducers produce the final outputs (FOs)
- Overall, MapReduce breaks the data flow into two phases, *map phase* and *reduce phase*



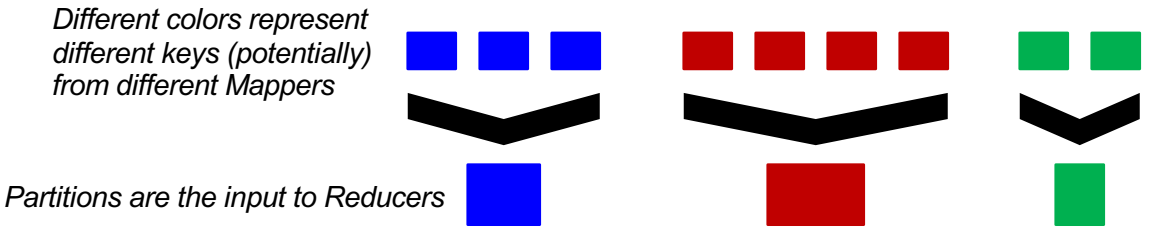
Keys and Values

- The programmer in MapReduce has to specify two functions, the *map function* and the *reduce function* that implement the Mapper and the Reducer in a MapReduce program
- In MapReduce data elements are always structured as key-value (i.e., (K, V)) pairs
- The map and reduce functions receive and *emit* (K, V) pairs

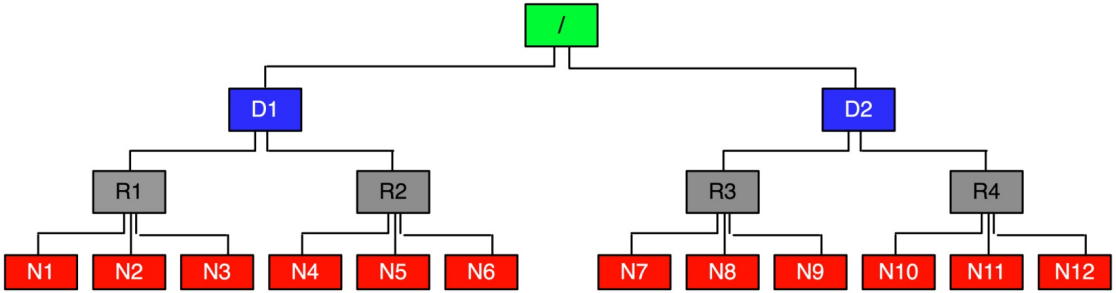


Partitions

- In MapReduce, intermediate output values are not usually reduced together
- *All values with the same key are presented to a single Reducer together*
- More specifically, a different subset of intermediate key space is assigned to each Reducer
- These subsets are known as *partitions*



Network Topology In MapReduce



- MapReduce assumes a tree style network topology
- Nodes are spread over different racks embraced in one or many data centers
- A salient point is that the bandwidth between two nodes is dependent on their relative locations in the network topology
- For example, nodes that are on the same rack will have higher bandwidth between them as opposed to nodes that are off-rack

MapReduce

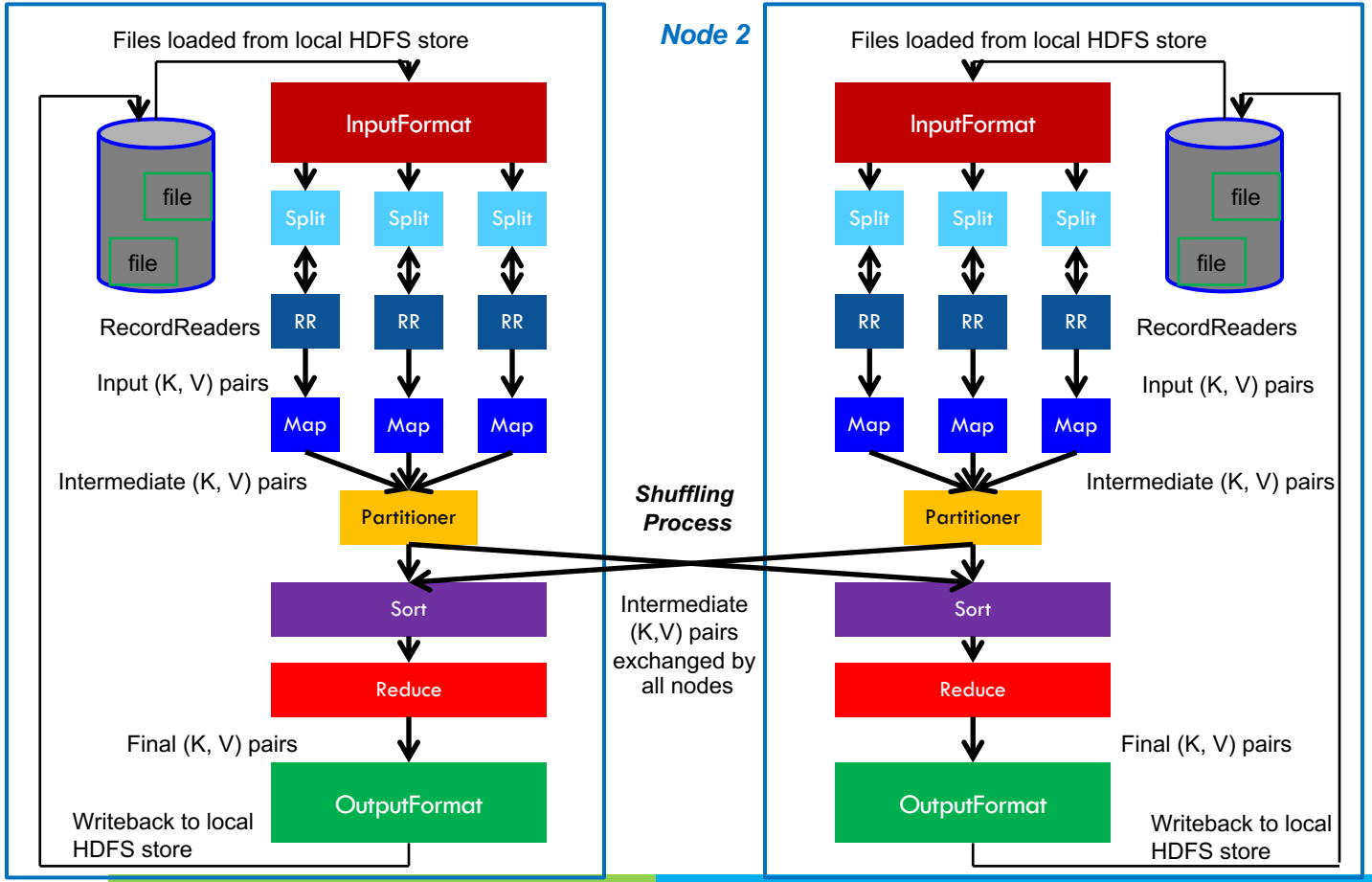
Concepts of MapReduce:

- Basics
- MapReduce data flow
- Additional functionality
- Scheduling and fault-tolerance in MapReduce
- Comparison with existing techniques and models

Hadoop

- Since its debut on the computing stage, MapReduce has frequently been associated with *Hadoop*
- Hadoop is an open source implementation of MapReduce and is currently enjoying wide popularity
- Hadoop presents MapReduce as an analytics engine and under the hood uses a distributed storage layer referred to as Hadoop Distributed File System (*HDFS*)
- HDFS mimics Google File System (*GFS*)

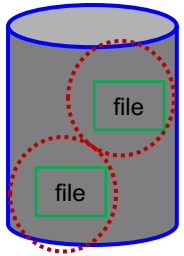
Hadoop MapReduce: A Closer Look



Input Files

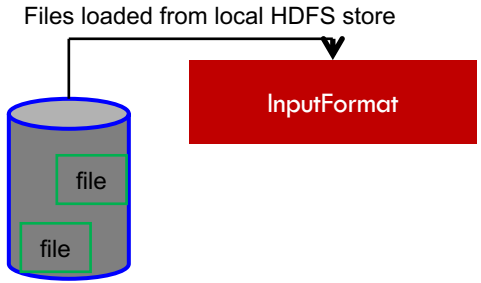
- *Input files* are where the data for a MapReduce task is initially stored
- The input files typically reside in a distributed file system (e.g. HDFS)

- The format of input files is arbitrary
  - Line-based log files
  - Binary files
  - Multi-line input records
  - Or something else entirely



InputFormat

- How the input files are split up and read is defined by the *InputFormat*
- InputFormat is a class that does the following:
- Selects the files that should be used for input
- Defines the *InputSplits* that break a file
- Provides a factory for *RecordReader* objects that read the file



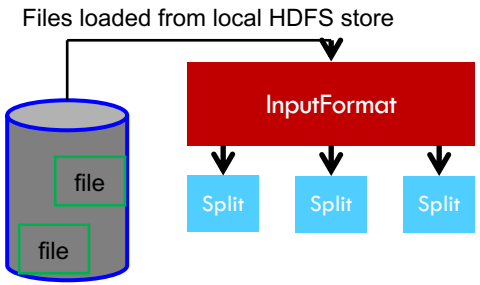
InputFormat Types

- Several InputFormats are provided with Hadoop:

InputFormat	Description	Key	Value
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into (K, V) pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

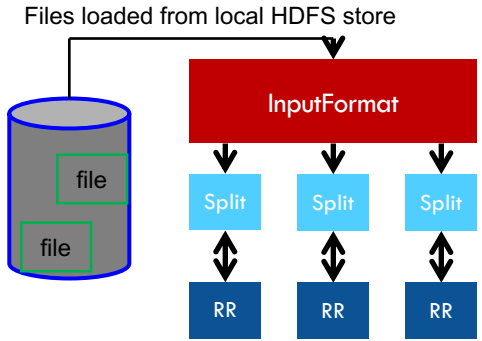
Input Splits

- An *input split* describes a unit of work that comprises a single map task in a MapReduce program
- the InputFormat breaks a file up into **splits (see default value)**
- By dividing the file into splits, we allow several map tasks to operate on a single file in parallel
- If the file is very large, this can improve performance significantly through parallelism
- **Each map task corresponds to a *single* input split**



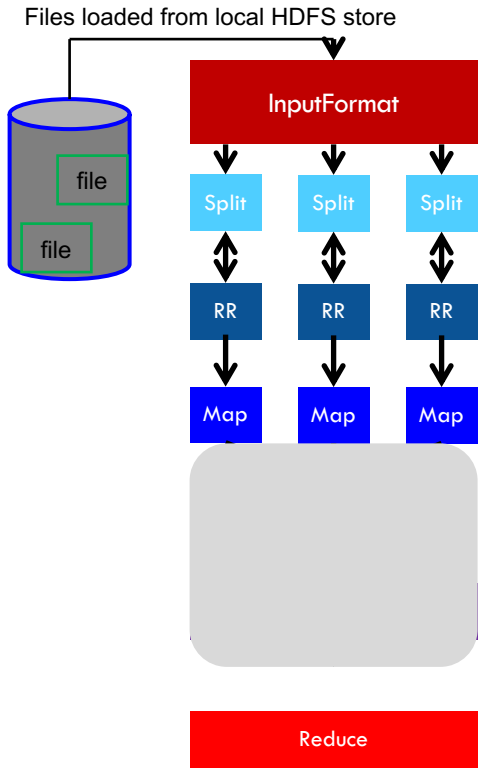
RecordReader

- The input split defines a slice of work but does not describe how to access it
- The *RecordReader* class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers
- The RecordReader is invoked repeatedly on the input until the entire split is consumed
- Each invocation of the RecordReader leads to another call of the map function defined by the programmer



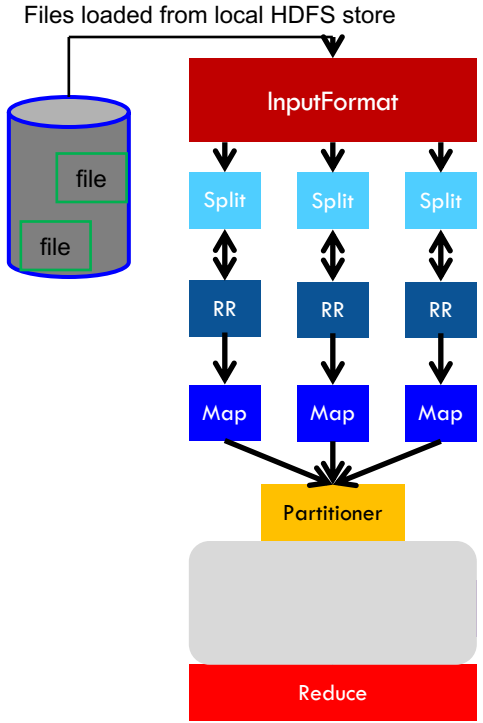
Mapper and Reducer

- The *Mapper* performs the user-defined work of the first phase of the MapReduce program
- A new instance of Mapper is created for each split
- The *Reducer* performs the user-defined work of the second phase of the MapReduce program
- A new instance of Reducer is created for each partition
- *For each key in the partition assigned to a Reducer, the Reducer is called once*



Partitioner

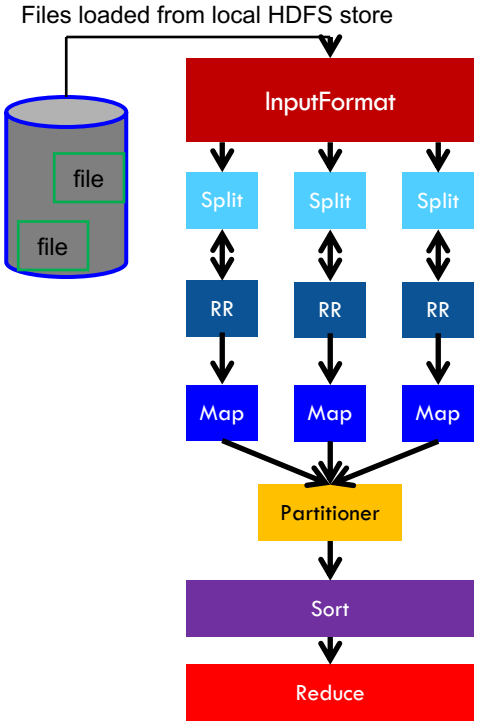
- Each mapper may emit (K, V) pairs to *any* partition
- Therefore, the map nodes must all agree on where to send different pieces of intermediate data
- The *partitioner* class determines which partition a given (K,V) pair will go to.
- The default partitioner computes *a hash value* for a given key and assigns it to a partition based on this result.



Sort

- Each Reducer is responsible for reducing the values associated with (several) intermediate keys

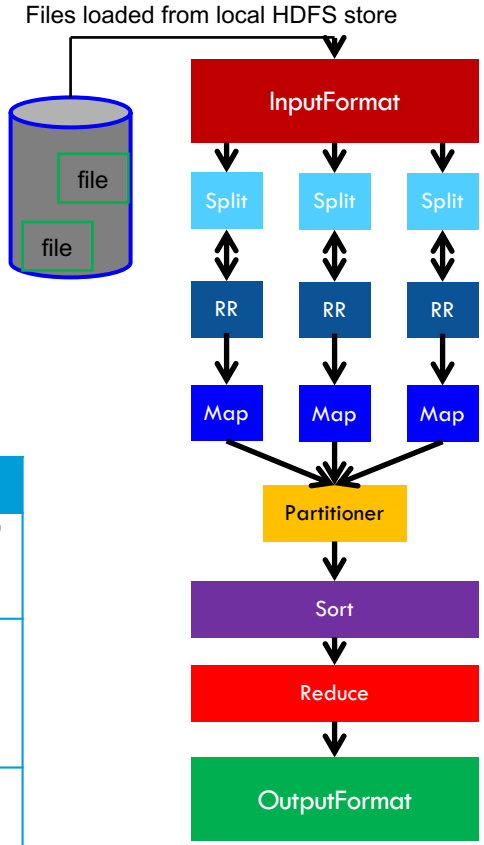
- The set of intermediate keys on a single node is *automatically sorted* by MapReduce before they are presented to the Reducer



OutputFormat

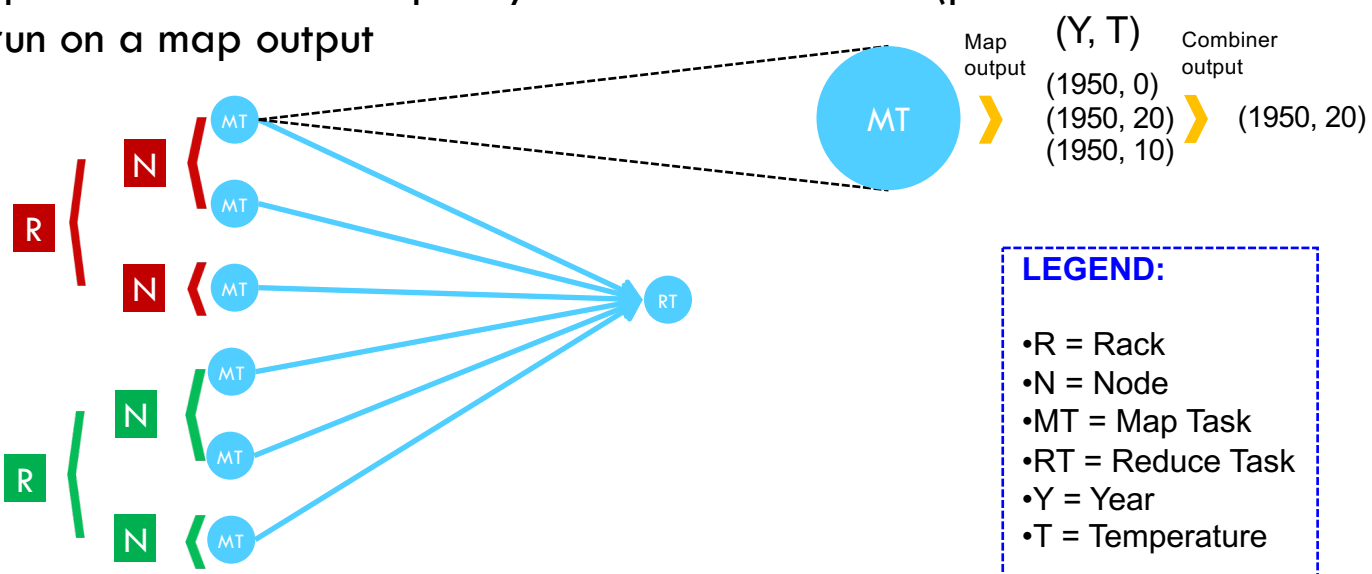
- The *OutputFormat* class defines the way (K,V) pairs produced by Reducers are written to output files
- The instances of OutputFormat provided by Hadoop write to files on the local disk or in HDFS
- Several OutputFormats are provided by Hadoop:

OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files



Combiner Functions

- MapReduce applications are limited by the bandwidth available on the cluster
- It pays to minimize the data shuffled between map and reduce tasks
- Hadoop allows the user to specify a combiner function (just like the reduce function) to be run on a map output

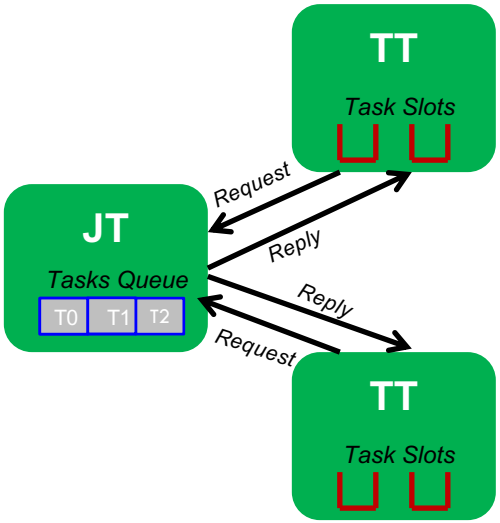


MapReduce

- Concepts of MapReduce described:
  - Basics
  - A close look at MapReduce data flow
  - Additional functionality
  - Scheduling and fault-tolerance in MapReduce

Task Scheduling in MapReduce

- MapReduce adopts a *master-slave architecture*
- The master node in MapReduce is referred to as *Job Tracker* (JT)
- Each slave node in MapReduce is referred to as *Task Tracker* (TT)



- MapReduce adopts a *pull scheduling* strategy rather than a *push one*
  - I.e., JT does not push map and reduce tasks to TTs but rather TTs pull them by making pertaining requests

Map and Reduce Task Scheduling

- Every TT sends a *heartbeat message* periodically to JT encompassing a request for a map or a reduce task to run

I. Map Task Scheduling:

   - JT satisfies requests for map tasks via attempting to schedule mappers in the *vicinity* of their input splits (i.e., it considers locality)

II. Reduce Task Scheduling:

   - However, JT simply assigns the next yet-to-run reduce task to a requesting TT regardless of TT's network location and its implied effect on the reducer's shuffle time (i.e., it does not consider locality)

Job Scheduling in MapReduce

- In MapReduce, an application is represented as a *job*
- A job encompasses multiple map and reduce tasks
- MapReduce in Hadoop comes with a choice of schedulers:
  - The default is the *FIFO scheduler* which schedules jobs in order of submission
  - There is also a multi-user scheduler called the *Fair scheduler* which aims to give every user a fair share of the cluster capacity over time

Fault Tolerance in Hadoop

- MapReduce can guide jobs toward a successful completion even when jobs are run on a large cluster where probability of failures increases
- The primary way that MapReduce achieves fault tolerance is through *restarting tasks*
- If a TT fails to communicate with JT for a period of time (by default, 1 minute in Hadoop), JT will assume that TT in question has crashed
  - If the job is still in the map phase, JT asks another TT to re-execute *all Mappers that previously ran at the failed TT*
  - If the job is in the reduce phase, JT asks another TT to re-execute *all Reducers that were in progress on the failed TT*

What Makes MapReduce Unique?

- MapReduce is characterized by:
  1. Its simplified programming model which allows the user to quickly write and test distributed systems
  2. Its efficient and automatic distribution of data and workload across machines
  3. Its flat scalability curve. Specifically, after a Mapreduce program is written and functioning on 10 nodes, very little-if any- work is required for making that same program run on 1000 nodes

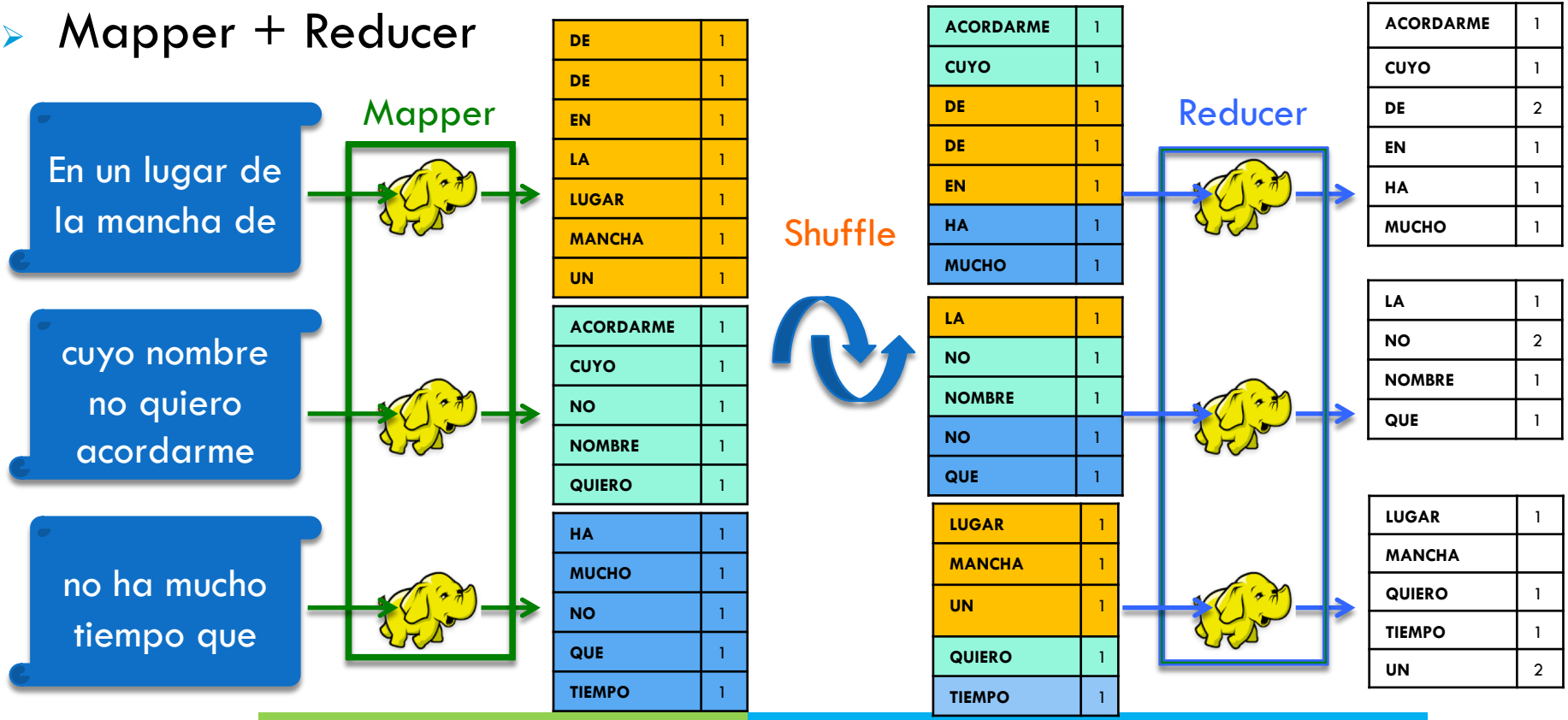
MapReduce: Insight

Consider the problem of counting the number of occurrences of each word in a large collection of documents

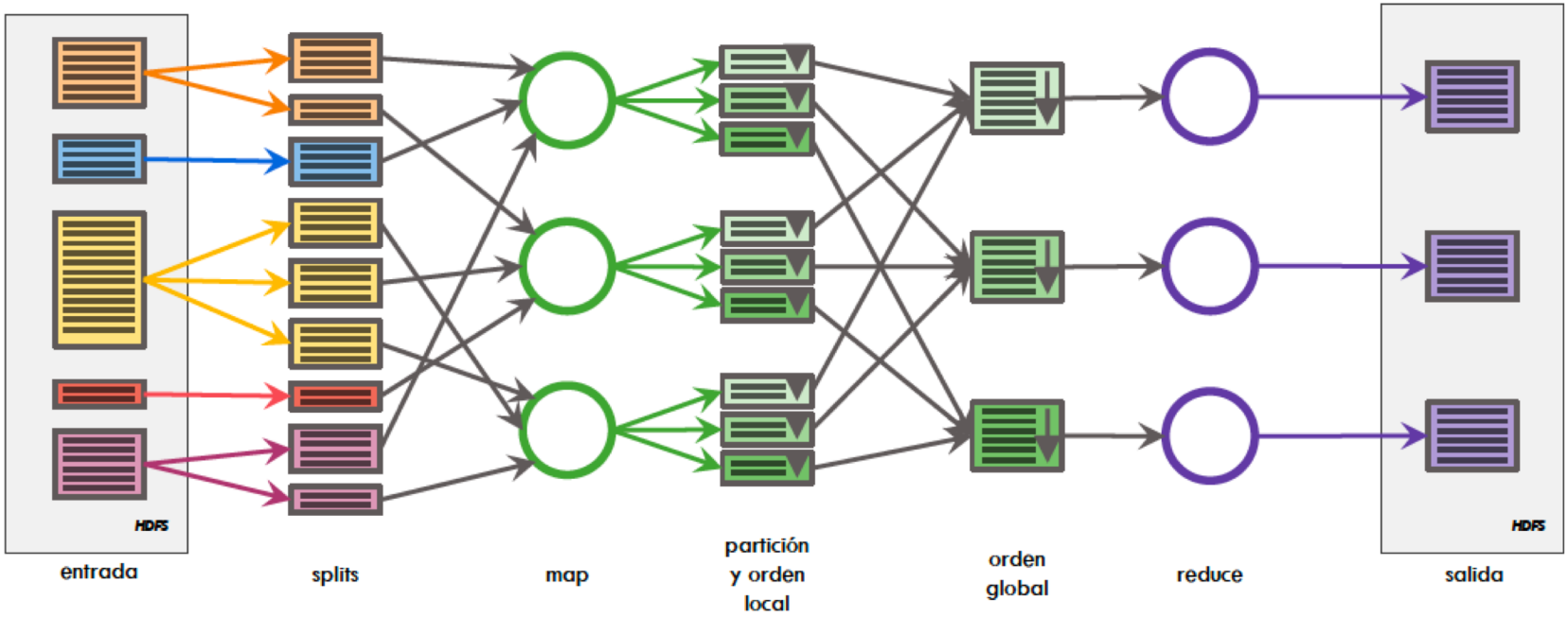
- How would you do it in parallel ?
- Solution:
  - Divide documents among workers
  - Each worker parses document to find all words, outputs (word, count) pairs
  - Partition (word, count) pairs across workers based on word
  - For each word at a worker, locally add up counts

WordCount Example

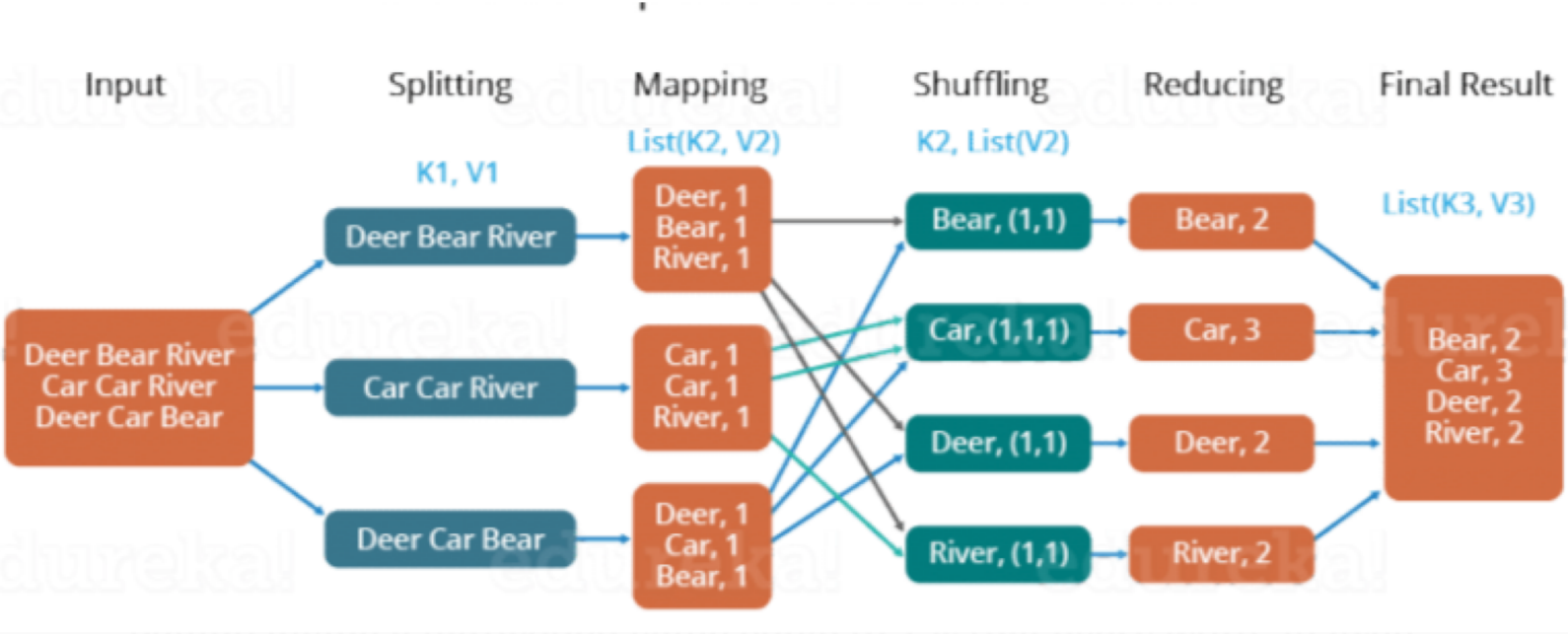
- Mapper + Reducer



WorkFlow: Map & Reduce



The Overall MapReduce Word Count Process



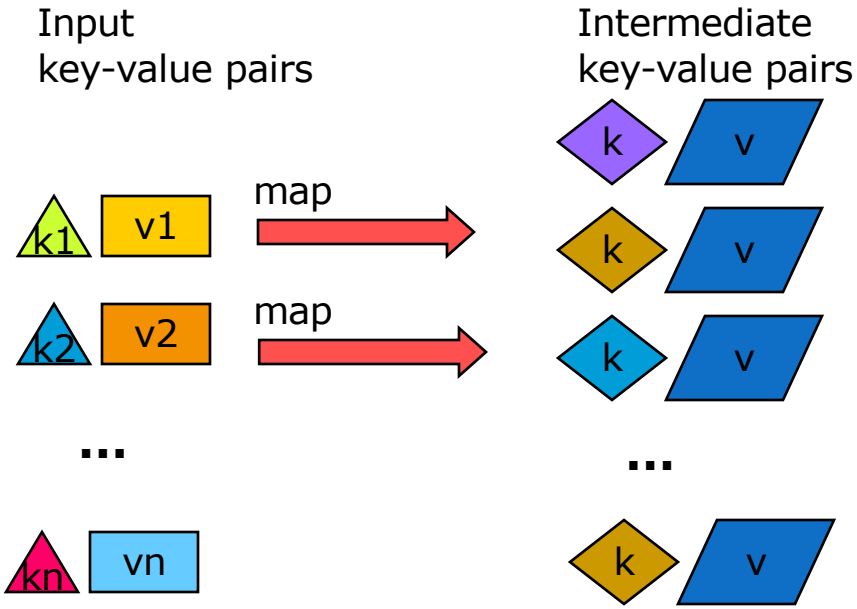
https://medium.com/edureka/mapreduce-tutorial-3d9535ddbe7c

MapReduce Programming Model

Inspired from map and reduce operations commonly used in functional programming languages like Lisp.

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs

MapReduce: The Map Step

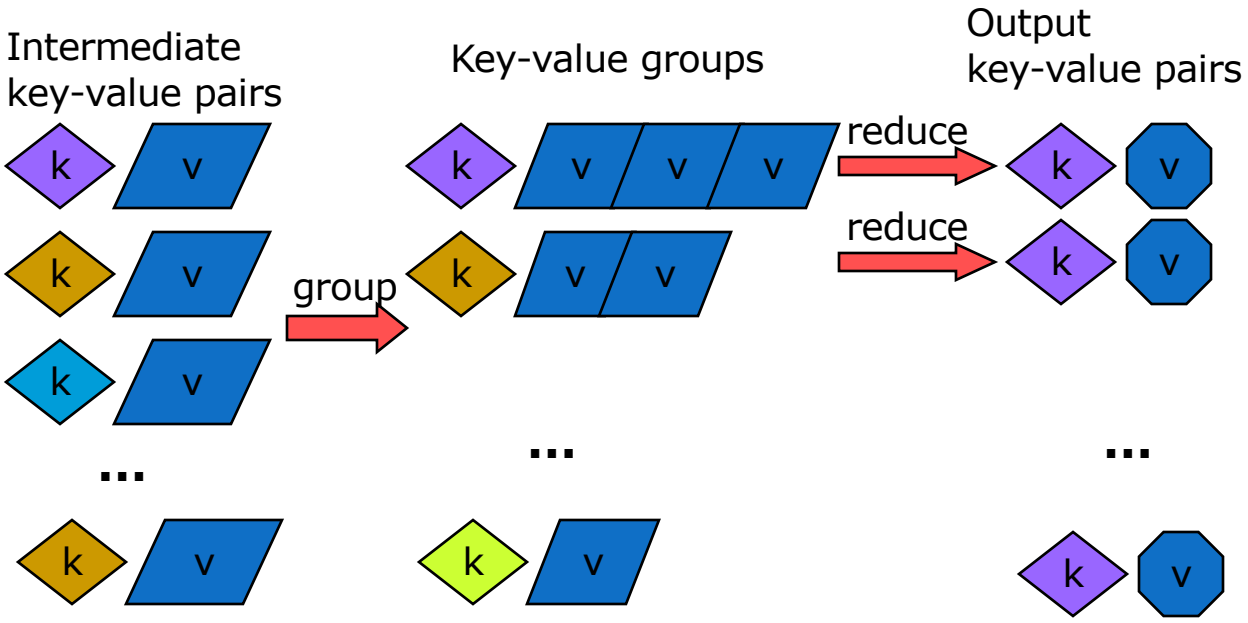


E.g. (doc—id, doc-content)

E.g. (word, wordcount-in-a-doc)

Adapted from Jeff Ullman's course slides

MapReduce: The Reduce Step



E.g.
(word, wordcount-in-a-doc)

(word, list-of-wordcount)
~ SQL Group by

(word, final-count)
~ SQL aggregation

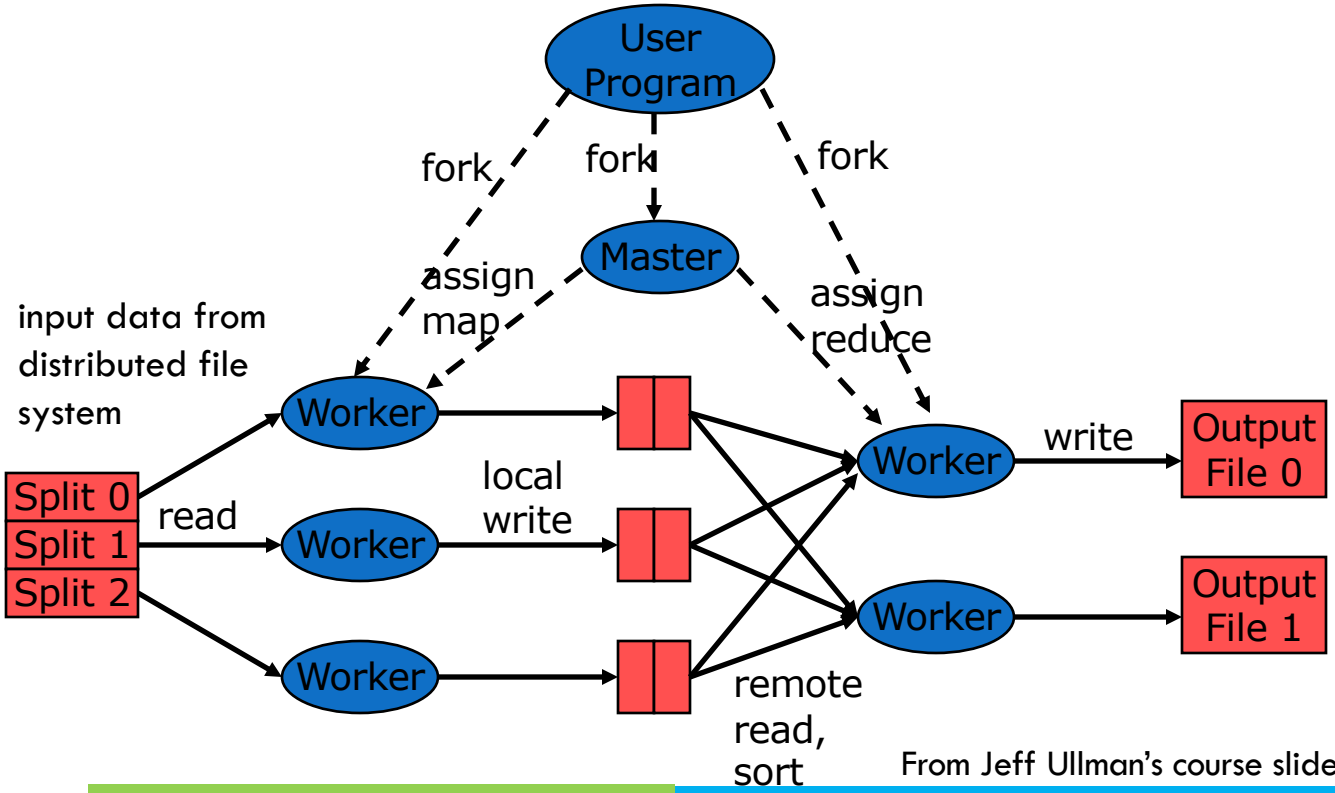
Adapted from Jeff Ullman's course slides

MapReduce

- MapReduce [OSDI'04] provides
  - Automatic parallelization, distribution
  - I/O scheduling
    - Load balancing
    - Network and data transfer optimization
  - Fault tolerance
    - Handling of machine failures
- **Need more power: Scale out, not up!**
  - Large number of **commodity servers** as opposed to some high end specialized servers

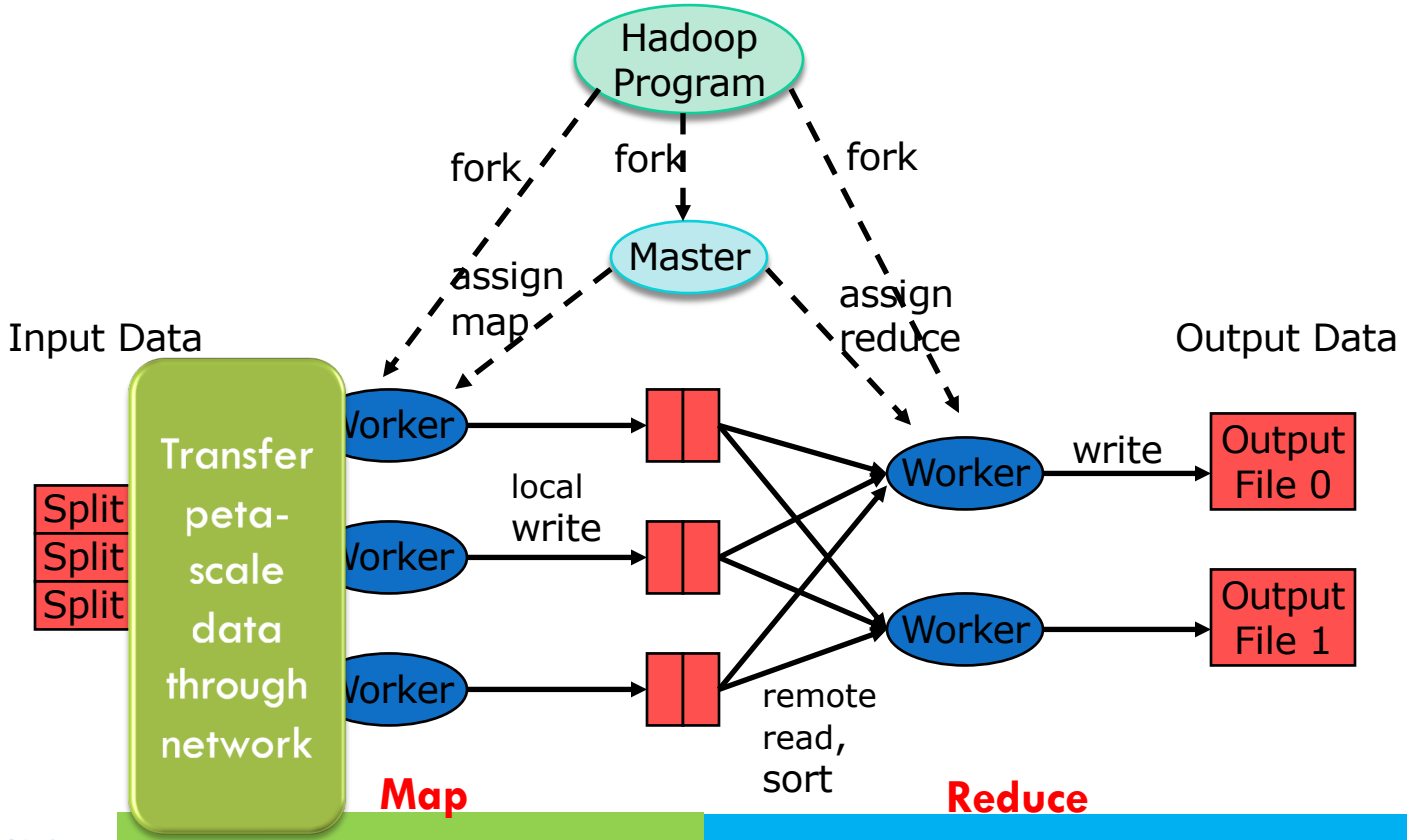
Apache Hadoop:
Open source implementation of MapReduce

Distributed Execution Overview

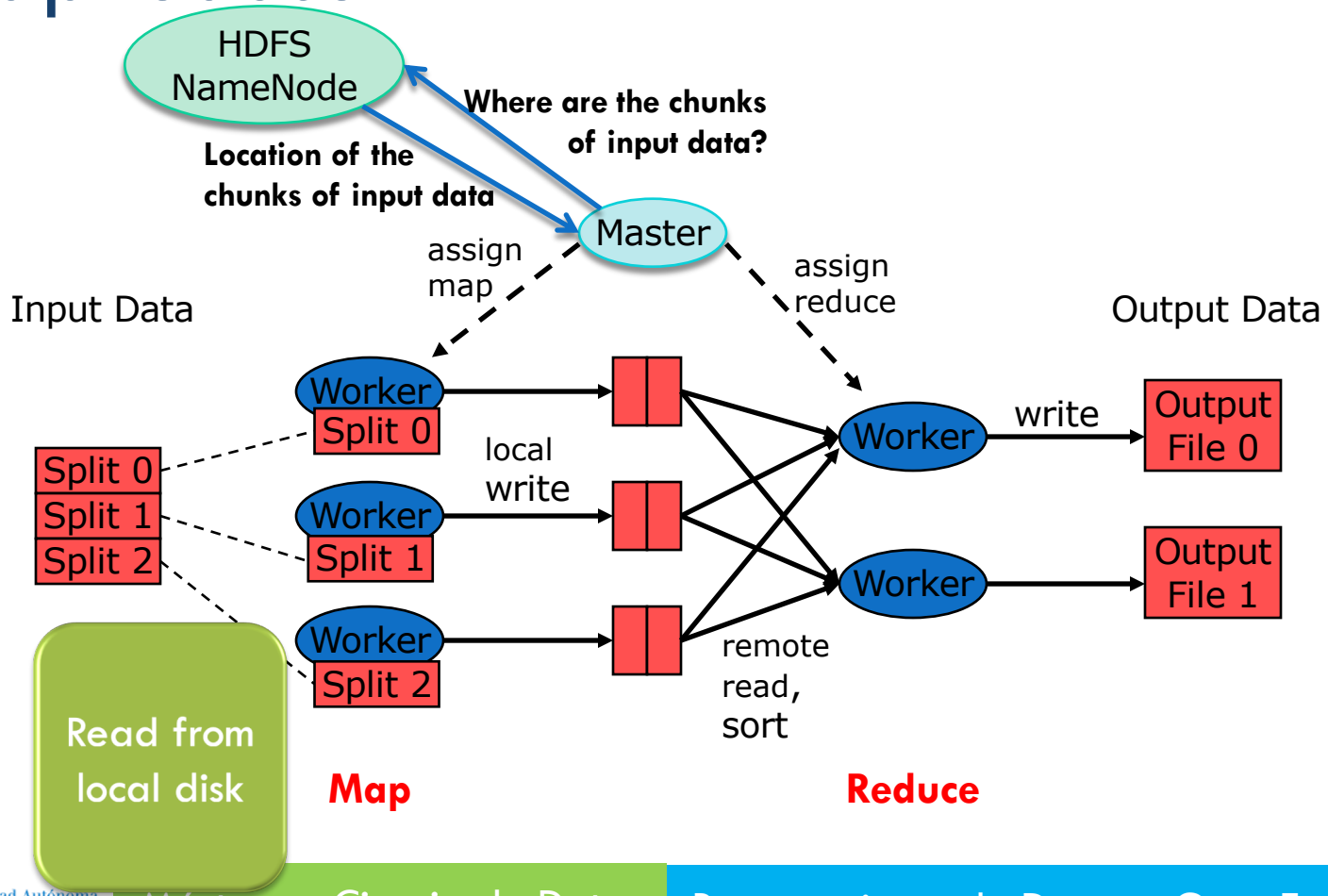


From Jeff Ullman's course slides

MapReduce


Hadoop Program
fork
fork
fork
Master
assign map
assign reduce
Input Data
Output Data
Split
Split
Split
Transfer peta-scale data through network
Worker
Worker
Worker
local write
Worker
Worker
write
Output File 0
Output File 1
remote read, sort
Map
Reduce

MapReduce


HDFS NameNode
Where are the chunks of input data?
Location of the chunks of input data
Master
assign map
assign reduce
Input Data
Output Data
Worker
Split 0
Worker
Split 1
Worker
Split 2
Split 0
Split 1
Split 2
local write
Worker
Worker
write
Output File 0
Output File 1
remote read, sort
Read from local disk
Map
Reduce

Hadoop: Components

Hadoop is an open-source software framework for distributed storage and processing of large datasets.

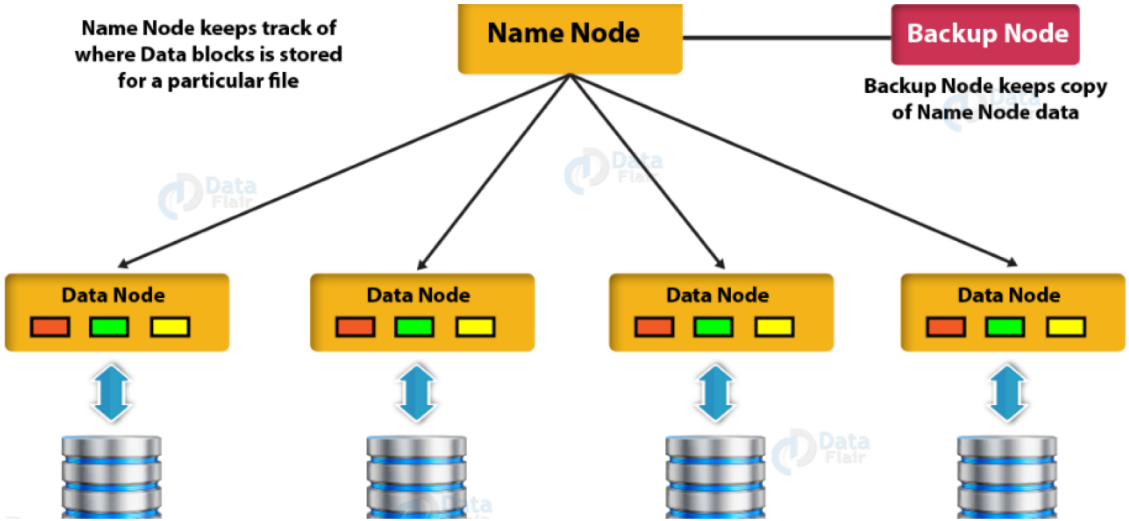
Apache Hadoop core components are:

- HDFS,
- MapReduce,
- and YARN.

Hadoop: Components

HDFS- Hadoop Distributed File System (HDFS) is the primary storage system of Hadoop.

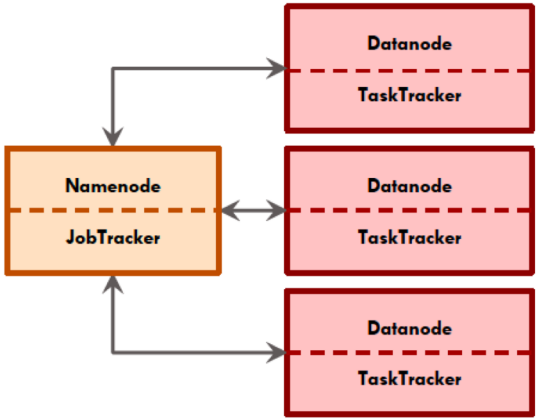
- HDFS stores data reliably even in the case of hardware failure.
- It provides high throughput access to an application by accessing in parallel.



Hadoop: Components

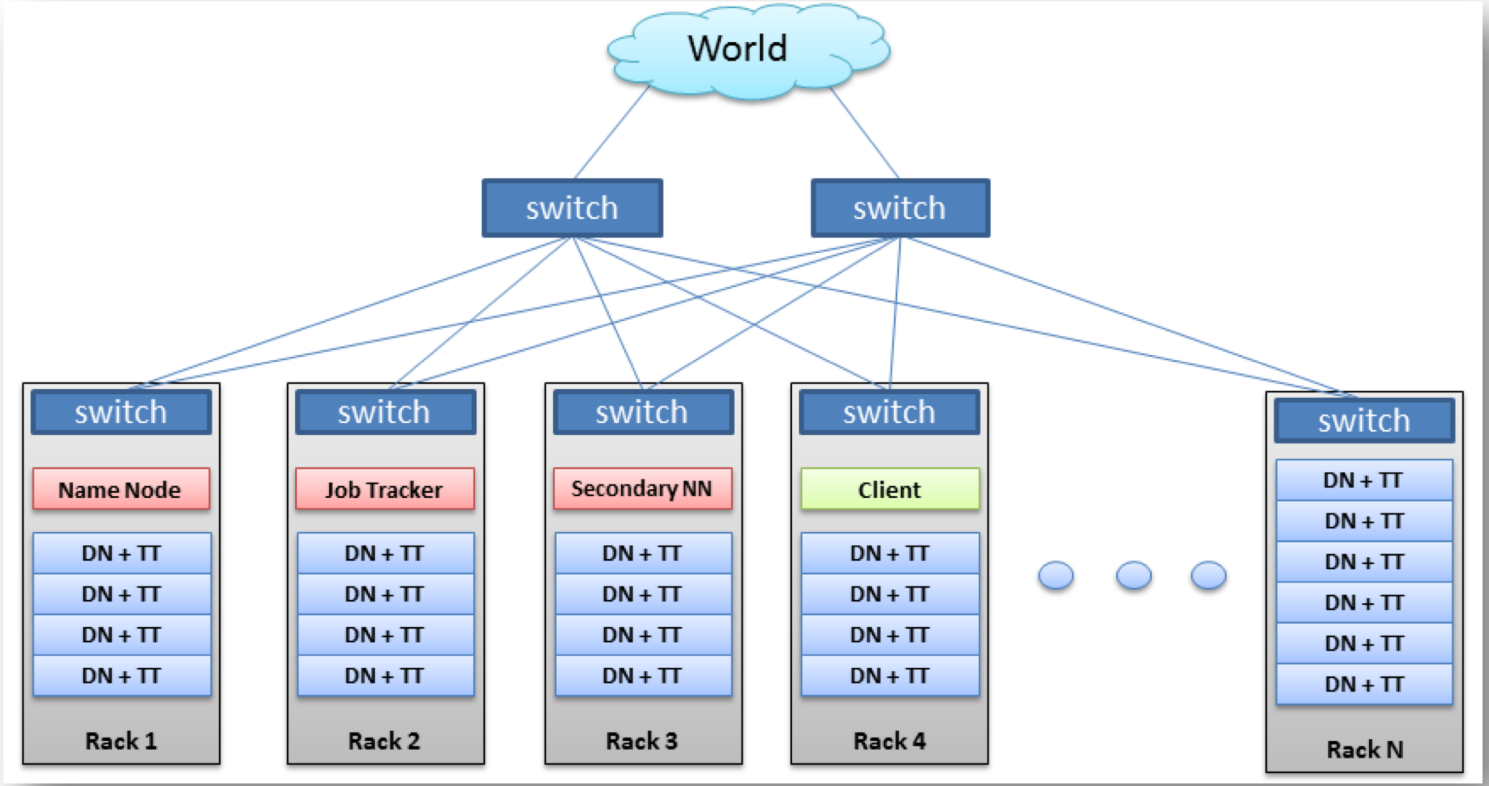
MapReduce- MapReduce is the data processing layer of Hadoop.

- It writes an application that processes large structured and unstructured data stored in HDFS.
- MapReduce processes a huge amount of data in parallel. It does this by dividing the job (submitted job) into a set of independent tasks (sub-job).
- In Hadoop, MapReduce works by breaking the processing into phases: **Map** and **Reduce.**
  - The Map is the first phase of processing, where we specify all the complex logic code.
  - Reduce is the second phase of processing. Here we specify light-weight processing like aggregation/summation.



YARN- YARN is the processing framework in Hadoop. It provides Resource management and allows multiple data processing engines. For example real-time streaming and batch processing.

Arquitectura Cluster Hadoop



CONCLUSION

Hadoop Map-Reduce is a software framework for easily writing **applications** which **process vast amounts of data** (multi-terabyte data-sets) **in-parallel** on large clusters (thousands of nodes) of **commodity** hardware in a **reliable, fault-tolerant manner.**